Instituto Tecnológico de Costa Rica

Departamento de Ing. Computación

Sede Cartago

Compiladores e Intérpretes (IC-5701)

Prof. Esteban Arias Méndez

Periodo II

26-09-2016

Tarea 4
Lex&Yacc + Micro

Carlos Adán Arguello Calderón
201173805

Luis Diego Flores Arguedas
201024948

Abstract:

Lex and Yacc are two useful tools in the process of creating parsers and scanners for recognition and construction of a language. The objective of this assignment is to use these tools and create an interpreter for the language MICRO.

Definición y descripción de herramientas de Lex & Yacc

1. Lex

Es un programa para generar analizadores léxicos. Lex se utiliza comúnmente con el programa Yacc que se utiliza para generar análisis sintáctico. Lex toma como entrada una especificación de analizador léxico y devuelve como salida el código fuente implementando el analizador léxico en C.

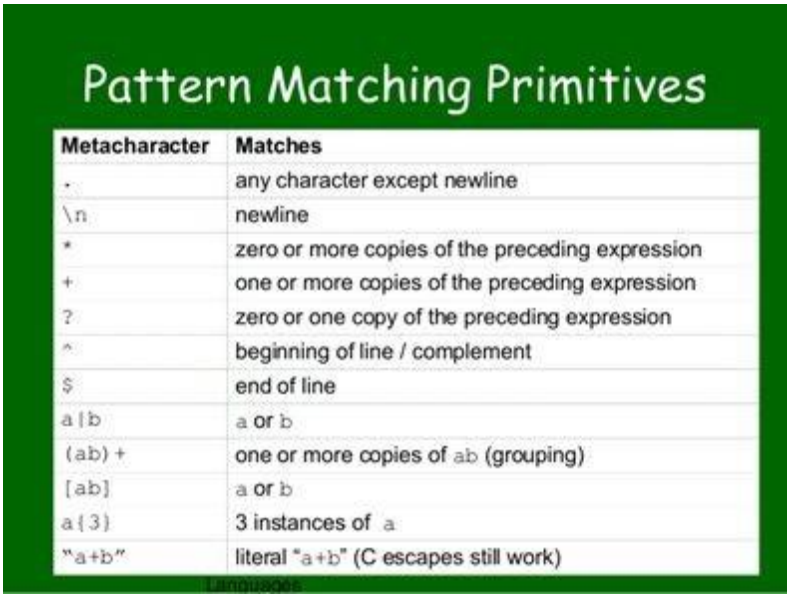
La estructura de un archivo Lex se hace de esta manera:

Sección de declaraciones
%%
Sección de reglas
%%
Sección de código C

- La **sección de declaraciones** es el lugar para definir macros y para importar los archivos de cabecera escritos en C. También es posible escribir cualquier código de C aquí, que será copiado en el archivo fuente generado. Este código en C debe ir entre los símbolos %{ %}.
- La **sección de reglas** es la sección más importante; asocia patrones a sentencias de C. Los patrones son simplemente expresiones regulares. Cuando el *lexer* encuentra un texto en la entrada que es asociable a un patrón dado, ejecuta el código asociado de C.
- La **sección de código C** contiene sentencias en C y funciones que serán copiadas en el archivo fuente generado. Estas sentencias contienen generalmente el código llamado por las reglas en la sección de las reglas. En programas grandes es más conveniente poner este código en un archivo separado y enlazarlo en tiempo de compilación.

Los siguientes cuadros ayudarán al trabajo de la creación del código Lex

1. Primitivas de coincidencia de patrones



2. Ejemplos de coincidencias de patrones

Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: `a`, `b`, `c`
`[a-z]`	any letter, a-z
`[a\-z]`	one of: `a`, `-`, `z`
`[-az]`	one of: `-`, `a`, `z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a`, `b`
`[a^b]`	one of: `a`, `^`, `b`
`[a|b]`	one of: `a`, `|`, `b`
`a|b`	one of: `a`, `b`

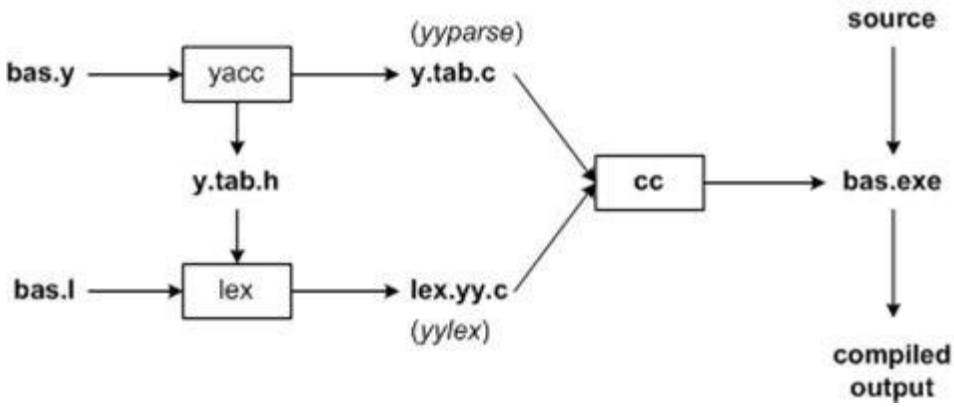
3- Variables predefinidas de Flex

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

2. Yacc

Programa informático de entrada general tiene algún tipo de estructura; de hecho, cada programa de computadora que hace de entrada puede ser pensado como la definición de un "idioma de entrada" aceptada por ésta. Un idioma de entrada puede ser tan complejo como un lenguaje de programación, o tan simple como una secuencia de números. Por desgracia, las instalaciones de entrada habituales son limitadas, difíciles de usar, y con frecuencia son laxos sobre la comprobación de sus entradas para la validez.

Yacc proporciona una herramienta general para la descripción de la entrada a un programa de ordenador. El usuario Yacc especifica las estructuras de su entrada, junto con el código que puede invocarse como se reconoce cada dicha estructura. Yacc convierte tal especificación en una subrutina que Han- DLE el proceso de entrada; Con frecuencia, es conveniente y apropiado tener la mayor parte del flujo de control en la aplicación del usuario a cargo de esta subrutina. A continuación,se presenta una imagen de todo el proceso que hace Lex & Yacc para generar el scanner y parser:



Las gramáticas para Yacc son una variante de la notación Backus-Naur (BNF), usadas para expresar lenguajes libres de contexto.

Muchos de los lenguajes modernos se pueden expresar en dicha notación. A continuación, se presenta un ejemplo de una expresión que multiplica y suma números:

E -> E + E

E -> E * E

E -> id

En el ejemplo anterior se denotan tres expresiones diferentes, los términos a la izquierda de la flecha como la expresión 'E' son denominado no terminales, ya que pueden producir los términos al lado derecho de la flecha. Lo términos como 'id' son expresiones terminales puesto que no generan ninguna otra expresión.

La gramática mostrada anteriormente muestra las expresiones que pueden ser suma de dos expresiones, producto de dos expresiones o un identificador respectivamente. A continuación, se muestra como la gramática genera la expresión requerida.

E -> E * E	(r2)
-> E * z	(r3)
-> E + E * z	(r1)
-> E + y * z	(r3)
-> x + y * z	(r3)

En cada paso se denota la "regla" de la gramática utilizada, expandiendo cada elemento no terminal hasta llegar a la expresión aritmética x+y*z.

Para "parsear" una expresión se debe llevar a cabo el proceso contrario, en lugar de de comenzar con una expresión no terminal y llegar a una expresión desde la gramática, se reducirá una expresión a un único elemento no terminal.

1	. x + y * z	Cambio
2	x . + y * z	(r3)
3	E . + y * z	Cambio
4	E + . y * z	Cambio
5	E + y . * z	(r3)
6	E + E . * z	Cambio
7	E + E * . z	Cambio
8	E + E * z .	(r3)
9	E + E * E .	(r2) ,multiplicar
10	E + E .	(r1), sumar
11	E .	aceptada

Nótese que en la línea 6 se pudo haber realizado la regla 1 en lugar de haber hecho un cambio en los elementos, sin embargo esto hubiera alterado la prioridad de la multiplicación ante la suma, dando así un resultado erróneo, a esto se le conoce como conflicto de shift-reduce.

La gramática es ambigua porque hay más de una manera de producirla. De la misma forma la expresión E -> E + E es ambigua por si misma, ya que puede tanto desarrollarse la expresión E de la izquierda como la del lado derecho.

Un error de tipo reduce reduce se da en la siguiente gramática:

E -> T

E -> id

T -> id

En este ejemplo se puede observar como se puede llegar a la terminal id desde T o desde E por lo que la gramática es ambigua.

3. Análisis de resultados y funcionamiento de la calculadora

Para el mejor uso de la calculadora se ha considerado realizar solo la calculadora #3 porque se debe a que esta está más completa y realiza más funcionalidades.

Solo se debe ubicar en la carpeta donde se encuentren los archivos en ubunto y ejecutar los comandos

bison -y -d calc3.y
flex calc3.l gcc -c
y.tab.c lex.yy.c
gcc y.tab.o lex.yy.o calc3a.c -o calc3a.exe gcc
y.tab.o lex.yy.o calc3b.c -o calc3b.exe gcc
y.tab.o lex.yy.o calc3g.c -o calc3g.exe

Línea de ejecución	Creada correctamente	Ocurrió un error
bison -y -d calc3.y	X	
flex calc3.l	X	
gcc -c y.tab.c lex.yy.c	X	
gcc y.tab.c lex.yy.o calc3a.c -o calc3a.exe	X, se creó el ejecutable satisfactoriamente	
gcc y.tab.c lex.yy.o calc3b.c -o calc3b.exe	X, se creó el ejecutable satisfactoriamente	
gcc y.tab.c lex.yy.o calc3g.c -o calc3g.exe	X, se creó el ejecutable satisfactoriamente	

4. Pruebas exitosas y fallidas de la calculadora

Archivo ejecutable	Prueba fallida	Prueba exitosa
calc3a.exe	5+1 esto genera un error de sintaxis	print 5+1; 6
calc3b.exe	print 5-4 syntax error	print 5-4; push 5 push 4 sub print
calc3g.exe	print 10+5 Error de sintaxis	print 10+5; Graph 0: print | | | [+] | |----| | | c(10) c(5)

Para el caso de este primer ejercicio no se tomaron códigos de otras fuentes para su funcionalidad.

5. Análisis de resultados y funcionamiento de la Micro

Archivo creado	Creada correctamente	Ocurrió un error
Lexer.flex	X	
Lexer.java	X	
Yylex.java	X	

6. Pruebas exitosas y fallidas de la Micro

7. Herramientas utilizadas

- Java
  Lenguaje multiplataforma utilizado para utilizar la herramienta Jflex.

- Jflex
  Analizador léxico escrito en java.

- GCC (Compilador de C)
  1. Abrir el terminal
  2. Escribir el siguiente comando para instalarlo: sudo-apt get install gcc

- Flex

  Para poder obtener esta herramienta se debe:
  1. Abrir el terminal
  2. En la terminal se escribe el siguiente comando: sudo apt-get install flex
  3. Y por último escribes tu contraseña de administrador para dar los permisos
- Bison

  Para poder obtener esta herramienta se debe:
  1. Abrir el terminal

2. En la terminal se escribe el siguiente comando: sudo apt-get install bison
3. Y por último escribes tu contraseña de administrador para dar los permisos

Se debe tener en cuenta que las herramientas instaladas anteriormente son para el uso de lex & yacc por lo tanto para su uso adecuado es importante saber:

Para compilar un archivo lex se debe:

1. Escribir el programa y guardarlo en extensión .l
2. Abrir el terminal y ubicarse en el directorio donde esta el archivo con extensión .l
3. Escribir el comando donde file es el nombre del archivo sería: lex file.l
4. Luego se debe poner la siguiente instrucción: cc lex.yy.c -ll
5. Para terminar se debe poner: ./a.out
6. Programa de lex corriendo exitosamente

Para compilar un archivo yacc se debe:

1. Escribir un programa lex en un archivo file.l y un yacc en un archivo file.y
2. Abrir el terminal y navegar al directorio donde se guardó el archivo
3. Escribir el comando: lex file.l
4. Escribir el siguiente comando: yacc file.y
5. Luego se debe escribir: cc lex.yy.c y.tab.h -ll
6. Para terminar se debe escribir: ./a.out
7. Para este último lex & yacc están corriendo satisfactoriamente; esto se debe a que yacc para poderlo compilar se debe primero compilar lex para su completa funcionalidad.

8. Apéndices

- Para el primer ejercicio se dispondrá de estos archivos fuentes

❖ Calc3.h

Para este archivo se definirá el tipo de estructura a utilizar

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */ typedef struct {     int value;
/* value of constant */ } conNodeType;

/* identifiers */ typedef struct {     int i;
/* subscript to sym array */ } idNodeType;

/* operators */ typedef
struct {
    int oper;                   /* operator */     int
nops;                   /* number of operands */
    struct nodeTypeTag *op[1];       /* operands, extended at runtime */ }
oprNodeType;
```

```
typedef struct nodeTypeTag {
    nodeEnum type;              /* type of node */
     union {          conNodeType con;        /*
constants */         idNodeType id;          /*
identifiers */         oprNodeType opr;
/* operators */
    };
} nodeType;
 extern int
sym[26];
```

- Calc3.l

Para este tipo de archivo de extensión .l se dispondrá tipos de funciones y expresiones regulares utilizados.

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h" void
yyerror(char *);
%}

%%

[a-z]       {                   yylval.sIndex =
*yytext - 'a';                  return
VARIABLE;
            }

0           {
                yylval.iValue = atoi(yytext);
return INTEGER;
            }

[1-9][0-9]* {
                yylval.iValue = atoi(yytext);
return INTEGER;
            }

[-()<>=+*/;{}.] {
                return *yytext;
             }

">="            return GE;
```

```
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]+        ;       /* ignore whitespace */

.               yyerror("Unknown character");
%% int
yywrap(void) {
return 1;
}
```

- Calc3.y

Acá es donde tiene toda la funcionalidad de la manera como se ira a interpretar el código

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i); nodeType *con(int
value); void freeNode(nodeType *p);
int ex(nodeType *p); int yylex(void);

void yyerror(char *s); int sym[26];
/* symbol table */
%}
 %union
{
    int iValue;                 /* integer value */
char sIndex;                /* symbol table index */
nodeType *nPtr;             /* node pointer */ };

%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list
```

```
%%  program:         function
{ exit(0); }
        ;
function:
          function stmt         { ex($2); freeNode($2); }
        | /* NULL */
        ;
stmt:
          ';'                           { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'                      { $$ = $1; }
        | PRINT expr ';'                { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'         { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}'             { $$ = $2; }
        ;
stmt_list:
          stmt                  { $$ = $1; }
        | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
        ;
expr:
          INTEGER               { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr         { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr         { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr         { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr         { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr         { $$ = opr('>', 2, $1, $3); }
        | expr GE expr          { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr          { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr          { $$ = opr(NE, 2, $1, $3); }
| expr EQ expr          { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'          { $$ = $2; }
        ;

%%
nodeType *con(int value) {
nodeType *p;

    /* allocate node */
    if ((p = malloc(sizeof(nodeType))) == NULL)
yyerror("out of memory");

    /* copy information */     p-
>type = typeCon;     p->con.value
= value;

    return p;
}  nodeType *id(int
i) {     nodeType
*p;

    /* allocate node */
    if ((p = malloc(sizeof(nodeType))) == NULL)
yyerror("out of memory");

    /* copy information */
p->type = typeId;     p->id.i
= i;

    return p;
```

```
}
nodeType *opr(int oper, int nops, ...) {
va_list ap;     nodeType *p;     int i;

    /* allocate node, extending op array */
    if ((p = malloc(sizeof(nodeType) + (nops-1) * sizeof(nodeType *))) == NULL)
yyerror("out of memory");

    /* copy information */
p->type = typeOpr;     p-
>opr.oper = oper;     p-
>opr.nops = nops;
va_start(ap, nops);     for
(i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
va_end(ap);     return p;
}
void freeNode(nodeType *p) {
int i;

    if (!p) return;     if (p->type ==
typeOpr) {         for (i = 0; i < p-
>opr.nops; i++)             freeNode(p-
>opr.op[i]);
    }
free (p);
}
void yyerror(char *s) {
fprintf(stdout, "%s\n", s);
}  int
main(void) {
yyparse();
return 0;
}
```

Para estos ver en el cuadro de pruebas exitosas y fallidas para entender mejor su concepto ❖

Calc3a.c

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"
 int ex(nodeType *p) {     if (!p) return 0;
switch(p->type) {     case typeCon:
return p->con.value;     case typeId:
return sym[p->id.i];     case typeOpr:
        switch(p->opr.oper) {         case WHILE:     while(ex(p-
>opr.op[0])) ex(p->opr.op[1]); return 0;         case IF:        if (ex(p-
>opr.op[0]))                              ex(p->opr.op[1]);
else if (p->opr.nops > 2)                              ex(p->opr.op[2]);
return 0;
        case PRINT:     printf("%d\n", ex(p->opr.op[0])); return 0;
case ';':       ex(p->opr.op[0]); return ex(p->opr.op[1]);         case
'=':       return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
case UMINUS:    return -ex(p->opr.op[0]);
        case '+':       return ex(p->opr.op[0]) + ex(p->opr.op[1]);
case '-':       return ex(p->opr.op[0]) - ex(p->opr.op[1]);         case
'*':       return ex(p->opr.op[0]) * ex(p->opr.op[1]);         case '/':
return ex(p->opr.op[0]) / ex(p->opr.op[1]);         case '<':
return ex(p->opr.op[0]) < ex(p->opr.op[1]);         case '>':
return ex(p->opr.op[0]) > ex(p->opr.op[1]);         case GE:
return ex(p->opr.op[0]) >= ex(p->opr.op[1]);         case LE:
return ex(p->opr.op[0]) <= ex(p->opr.op[1]);         case NE:
return ex(p->opr.op[0]) != ex(p->opr.op[1]);         case EQ:
return ex(p->opr.op[0]) == ex(p->opr.op[1]);
        }
}     return
0;
}
```

- Calc3b.c

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

static int lbl;
 int ex(nodeType *p)
{     int lbl1,
lbl2;
     if (!p) return
0;     switch(p-
>type) {     case
typeCon:
        printf("\tpush\t%d\n", p->con.value);
break;     case typeId:
        printf("\tpush\t%c\n", p->id.i + 'a');
break;     case typeOpr:
        switch(p->opr.oper) {
case WHILE:
            printf("L%03d:\n", lbl1 = lbl++);
ex(p->opr.op[0]);
printf("\tjz\tL%03d\n", lbl2 = lbl++);
ex(p->opr.op[1]);
printf("\tjmp\tL%03d\n", lbl1);
printf("L%03d:\n", lbl2);             break;
case IF:
            ex(p->opr.op[0]);
if (p->opr.nops > 2) {
/* if else */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl2 = lbl++);
printf("L%03d:\n", lbl1);                  ex(p-
>opr.op[2]);                 printf("L%03d:\n", lbl2);
            } else {
/* if */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
ex(p->opr.op[1]);                 printf("L%03d:\n",
lbl1);
            }             break;
case PRINT:
ex(p->opr.op[0]);
printf("\tprint\n");
break;         case '=':
            ex(p->opr.op[1]);
printf("\tpop\t%c\n", p->opr.op[0]->id.i + 'a');
break;         case UMINUS:
            ex(p->opr.op[0]);
printf("\tneg\n");
break;         default:
            ex(p->opr.op[0]);             ex(p-
>opr.op[1]);             switch(p->opr.oper) {
case '+':   printf("\tadd\n"); break;
case '-':   printf("\tsub\n"); break;
case '*':   printf("\tmul\n"); break;
case '/':   printf("\tdiv\n"); break;
case '<':   printf("\tcompLT\n"); break;
case '>':   printf("\tcompGT\n"); break;
case GE:    printf("\tcompGE\n"); break;
case LE:    printf("\tcompLE\n"); break;
case NE:    printf("\tcompNE\n"); break;
case EQ:    printf("\tcompEQ\n"); break;
            }
        }
}     return
0;
```

```
}
```

- Calc3g.c

```
/* source code courtesy of Frank Thomas Braun */

/* calc3d.c: Generation of the graph of the syntax tree */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "calc3.h"
#include "y.tab.h"
 int del = 1; /* distance of graph columns
*/ int eps = 3; /* distance of graph lines
*/

/* interface for drawing (can be replaced by "real" graphic using GD or other)
*/ void graphInit (void); void graphFinish();
void graphBox (char *s, int *w, int *h); void
graphDrawBox (char *s, int c, int l); void
graphDrawArrow (int c1, int l1, int c2, int l2);

/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);
/*****************************************************************************/
/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p) {     int rte, rtm;

    graphInit ();     exNode (p,
0, 0, &rte, &rtm);
graphFinish();     return 0;
}

/*c----cm---ce---->                       drawing of leaf-nodes  l
leaf-info
 */

/*c---------------cm--------------ce----> drawing of non-leaf-nodes  l
node-info
*                |
*                -------------     ...----
*                |       |               |
*                v       v               v
*                child1  child2  ...     child-n
*                che     che             che
 *cs      cs      cs              cs
 *
 */  void exNode
(   nodeType *p,
        int c, int l,        /* start column and line of node */
int *ce, int *cm     /* resulting end column and mid of node */     )
{
    int w, h;           /* node width and height */
char *s;            /* node text */
    int cbar;           /* "real" start column of node (centred above subnodes) */
int k;              /* child number */
    int che, chm;       /* end column and mid of children */
int cs;             /* start column of children */     char
word[20];      /* extended node text */

    if (!p) return;
```

```
    strcpy (word, "???"); /* should never appear */     s = word;
switch(p->type) {         case typeCon: sprintf (word, "c(%d)", p-
>con.value); break;         case typeId:  sprintf (word, "id(%c)",
p->id.i + 'A'); break;         case typeOpr:
            switch(p->opr.oper){                 case
WHILE:     s = "while"; break;                 case
IF:        s = "if";    break;                 case
PRINT:     s = "print"; break;                 case
';':       s = "[;]";     break;
case '=':       s = "[=]";     break;
case UMINUS:    s = "[_]";     break;
case '+':       s = "[+]";     break;
case '-':       s = "[-]";     break;
case '*':       s = "[*]";     break;
case '/':       s = "[/]";     break;
case '<':       s = "[<]";     break;
case '>':       s = "[>]";     break;
case GE:        s = "[>=]";    break;
case LE:        s = "[<=]";    break;
case NE:        s = "[!=]";    break;
case EQ:        s = "[==]";    break;             }
            break;
    }

    /* construct node text box */
graphBox (s, &w, &h);     cbar =
c;     *ce = c + w;
    *cm = c + w / 2;

    /* node is leaf */
    if (p->type == typeCon || p->type == typeId || p->opr.nops == 0)
{         graphDrawBox (s, cbar, l);         return;
    }

    /* node has children */
cs = c;
    for (k = 0; k < p->opr.nops; k++) {         exNode
(p->opr.op[k], cs, l+h+eps, &che, &chm);         cs =
che;
    }

    /* total node width */
if (w < che - c) {
        cbar += (che - c - w) / 2;
        *ce = che;
*cm = (c + che) / 2;
    }

    /* draw node */
graphDrawBox (s, cbar, l);

    /* draw arrows (not optimal: children are drawn a second time) */
cs = c;
    for (k = 0; k < p->opr.nops; k++) {         exNode
(p->opr.op[k], cs, l+h+eps, &che, &chm);
graphDrawArrow (*cm, l+h, chm, l+h+eps-1);         cs
= che;
    }
}

/* interface for drawing */

#define lmax 200
#define cmax 200

char graph[lmax][cmax]; /* array for ASCII-Graphic */ int
graphNumber = 0;
```

```
void graphTest (int l, int c)
{   int ok;     ok = 1;     if (l < 0) ok = 0;     if (l >= lmax) ok = 0;     if
(c < 0) ok = 0;      if (c >= cmax) ok = 0;      if (ok) return;      printf
("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ... %d, %d",          l,
c, lmax, cmax);     exit(1);
}

void graphInit (void) {     int i,
j;     for (i = 0; i < lmax; i++) {
for (j = 0; j < cmax; j++) {
graph[i][j] = ' ';
        }
    }
}  void
graphFinish() {
int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
graph[i][cmax-1] = 0;         if (j < cmax-1)
graph[i][j+1] = 0;         if (graph[i][j] == ' ')
graph[i][j] = 0;
    }     for (i = lmax-1; i > 0 && graph[i][0] == 0;
i--);     printf ("\n\nGraph %d:\n", graphNumber++);
for (j = 0; j <= i; j++) printf ("\n%s", graph[j]);
printf("\n");
}
void graphBox (char *s, int *w, int *h) {
    *w = strlen (s) + del;
    *h = 1;
}
void graphDrawBox (char *s, int c, int l) {
int i;
    graphTest (l, c+strlen(s)-1+del);
for (i = 0; i < strlen (s); i++) {
graph[l][c+i+del] = s[i];
    }
}
void graphDrawArrow (int c1, int l1, int c2, int l2) {
int m;     graphTest (l1, c1);     graphTest (l2,
c2);     m = (l1 + l2) / 2;
    while (l1 != m) { graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--; }
while (c1 != c2) { graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--; }     while
(l1 != l2) { graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--; }
graph[l1][c1] = '|';
}
```

- Para el segundoejercicio se dispondrá de estos archivos fuentes

  - Lexer.flex
    Analizador léxico donde va acompañado con sus expresiones regulares y reglas.

    %line
    %full
    %char
    %ignorecase
    %class Yylex

    %eofval{

```
  return new Symbol(sym.EOF);
%eofval}

D = [0-9]
ID = [a-zA-Z]
WHITE = [ ]
WHITEN = [\n]
WHITET = [\t]
WHITER = [\r]
%{
public String lexico = "";
public static int linea=1;
public static int pos=0;
public bool COM = false;
%}
%%

[\t\r ]+ {lexico += " ";}
[\n] {COM = false;}
[B][E][G][I][N] {if(COM == false){{lexico+=yytext();return new
Symbol(sym.BEGIN);}else{lexico+=yytext();}}}
[E][N][D] {if(COM == false){{lexico+=yytext();return new
Symbol(sym.END);}else{lexico+=yytext();}}}
[R][E][A][D] {if(COM == false){{lexico+=yytext();return new
Symbol(sym.READ);}else{lexico+=yytext();}}}
[W][R][I][T][E] {if(COM == false){{lexico+=yytext();return new
Symbol(sym.WRITE);}else{lexico+=yytext();}}}

[+-]?({D})* {if(COM == false){lexico=yytext();return new
Symbol(sym.INT);}else{lexico+=yytext();}}
":=" {if(COM == false){lexico += yytext(); return new
Symbol(sym.ASIGNA;)}else{lexico += yytext();}}
{ID}({ID})* {if(COM == false){lexico += yytext();lexico+=yytext();return new
Symbol(sym.IDENTIFICADOR);}else{lexico+=yytext();}}
";" {if(COM ==false){{lexico+=yytext();return new
Symbol(sym.PUNTO_COMA);}else{lexico+=yytext();}}}
"--" {{COM = true; {lexico+=yytext();return new
Symbol(sym.COMETARIO);}else{lexico+=yytext();}}}
```

- Lexer.Java
  Invocación del archivo flex.

```
package lexer;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
import java.io.*;
import java_cup.runtime.*;
import java.lang.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java_cup.*;

public class Lexer {

    public static void main(String[] args)
    {
        String _path = "/home/luis/NetBeansProjects/Lexer/src/lexer/Lexer.flex";
        String _pathtwo =
"/home/luis/NetBeansProjects/Lexer/src/lexer/Parser.cup";
        Lexereador(_path,_pathtwo);
        String dirName = null;
    }

    public static void Lexereador(String path, String pathtwo)
    {
        try
        {
            File _file = new File(path);
            jflex.Main.generate(_file);
            /*File _filetwo = new File(pathtwo);
            String[] _filetoo = new String[] {pathtwo};
            java_cup.Main.main(_filetoo);*/

        }
        catch(Exception e1)
        {
            System.out.println(e1);
        }
    }

}
```

9. Conclusiones

Las herramientas proporcionadas para la generación lenguajes disminuyen la dificultad para crear lenguajes diferentes; con respecto a la dificultad la disminuye porque se genera código que no se tiene que escribir automáticamente.

10. Observaciones finales

Para el primer ejercicio no hubo mucha dificultad en su elaboración; compiló bien y se ejecutó bien, acá la única observación es manejar C y la terminal de ubuntu.

11. Bibliografía

- http://dinosaur.compilertools.net Un poco de introducción a lex & yacc

- http://epaperpress.com/lexandyacc/

  Página dada por el profesor acá se encuentra el manual en pdf y el ejemplo de la calculadora

- http://www.embedded.com/design/prototyping-and-development/4024523/Lex-and-Yacc-for-Embedded-Programmers

  Explicaciones del Uso de lex & yacc

- http://easywaysnow.blogspot.com/2012/10/running-lex-and-yacc-program-in-ubuntu.html

  Como correr e instalar programas fuentes en lex y yacc

- http://askubuntu.com/questions/154402/install-gcc-on-ubuntu-12-04-lts

  Instalación del compilador GCC de C